Lattice QCD GPU programming with Quda and Grid

November 12, 2025

renwick.james.hudspith@gmail.com

Outline

- slaph interface in QUDA
  - meson functions
  - Baryon kernel

Recap on the last lecture

We learned that there is an interface for *GEMM routines in QUDA and it will translate automatically between row-major and column-major formats. The interface is shallow and it will be needed for the rest of this lecture.

There is no standard batched BLAS function call - MKL differs from cuBLAS and HipBLAS although the latter agree.

We introduced compilation and running with GRID, we talked a bit about the structure of GRID and how that translates to effective parallelism.

We saw that GRID hides the GPU interface from us but we can (and often should) drop down and use these "`accelerator_for`" loops, as substantial speedup is possible.

Meson currents

I showed the inner loop of the meson contractions in the last lecture, and teased it in the second. To recap we want to implement N DFTs as a small matrix batched with several right-hand-side larger matrices.

$$N_P\left\{\overbrace{\begin{bmatrix} e^{ip_0\cdot x_0} & \cdots & e^{ip_0\cdot x_{L^3}} \\ \vdots & \vdots & \vdots \\ e^{ip_{N_P-1}\cdot x_0} & \cdots & e^{ip_{N_P-1}\cdot x_{L^3}} \end{bmatrix}}^{L^3}\right. \times L^3\left\{\overbrace{\begin{bmatrix} Tr(x_0,t_0) & \cdots & Tr(x_0,t_{L_t-1}) \\ \vdots & \vdots & \vdots \\ Tr(x_{L^3},t_0) & \cdots & Tr(x_{L^3},t_{L_t-1}) \end{bmatrix}}^{T}\right.$$

Where the "N" is our "blockSizeMomProj" - a division of the $n1 \times n2$ noise possibilities we will run through.

What is the code doing? Well it is performing a DFT over traces of eigenvectors

$$\phi(i,j,p,t) = \sum_x e^{ip\cdot x} \mathrm{Tr}_C \left[v_i^\dagger(x,t) v_j(x,t) \right] \tag{1}$$

Where Tr_C is a trace over color indices and v are our Laplacian eigenvectors. This is a building block (MesonDoublet in the code) for forming correlators in combination with the perambulator.

Another useful building block is built from $\eta_i = \rho_{ij} v_j$, where ρ are our dilution noises in the laph subspace. We then replace v with η in the above equation (MesonKernel in the code).

Note that i moves slowest in this construction and t fastest. The innerProd color trace is just the kernel we created in lecture 2.

As this is an interface we expect the "latticeColorVectors" to be on the host. The function we will call is:

```
void laphMesonKernel( const int n1, const int n2,
                const int nMom,
                const int blockSizeMomProj,
                void **host_quark,
                void **host_quark_bar,
                const double _Complex *host_mom,
                QudaInvertParam inv_param,
                void *return_array,
                const int X[4])
```

All arguments here exist on the host. Here `"host_mom"` is a flattened array of $Np \times L^3$ Fourier twiddle factors. $X[4]$ carries the local dimensions.

Allocations

Here we allocate memory on the device, skipping creating the CPU colorSpinorField creation

```
const size_t data_ret_bytes = nMom*X[3]*n1*n2*2*precision;
const size_t data_tmp_bytes = n_sites*blockSizeMomProj*\
  2*precision;
const size_t data_mom_bytes = nMom*n_spatial_sites*\
  2*precision;
void *d_ret = pool_device_malloc(data_ret_bytes);
void *d_tmp = pool_device_malloc(data_tmp_bytes);
void *d_mom = pool_device_malloc(data_mom_bytes);
```

Here we need to specify the return array size for the device "`d_ret`" a temporary that holds the lattice-wide traces "`d_tmp`" and some space for the copied momentum twiddles "`d_mom`"

batched-strided ZGEMM parameters

We call the constructor

```
QudaBLASParam cublas_param_mom_sum = newQudaBLASParam();
```

And specify that the matrix to the right is transposed as our fields are in c-order

```
  cublas_param_mom_sum.trans_a = QUDA_BLAS_OP_N;
  cublas_param_mom_sum.trans_b = QUDA_BLAS_OP_T;
  cublas_param_mom_sum.m = nMom ;
  cublas_param_mom_sum.n = X[3] ;
  cublas_param_mom_sum.k   = n_spatial_sites ;
  cublas_param_mom_sum.lda = n_spatial_sites ;
  cublas_param_mom_sum.ldb = n_spatial_sites ;
  cublas_param_mom_sum.ldc = X[3] ;
```

"k" is our inner index for the matrix multiply and we will be creating an nMom $\times$ $X[3]$ output matrix.

BLAS params continued

Here we are saying that there will be no striding (i.e. we reuse the same matrix) in the left matrix of Fourier twiddles

```
cublas_param_mom_sum.a_stride = 0 ; // mom stays the same
cublas_param_mom_sum.b_stride = n_spatial_sites*X[3] ;
cublas_param_mom_sum.c_stride = X[3]*nMom ;
cublas_param_mom_sum.batch_count = blockSizeMomProj ;
```

And we will be batching the DFT over `blockSizeMomProj`.

```
cublas_param_mom_sum.alpha = 1.0;
cublas_param_mom_sum.beta = 0.0;
cublas_param_mom_sum.data_order = QUDA_BLAS_DATAORDER_ROW;
cublas_param_mom_sum.data_type = QUDA_BLAS_DATATYPE_Z;
```

Here we specify that the ZGEMM we want is all in row-major format

We start some copies. If doing a MesonDoublet we just copy all the evecs to the device although this could be tiled. For a MesonKernel we call this function:

```
const size_t nEv = evec.size() ;
std::vector<ColorSpinorField> quda_evec(1) ;
quda_evec[0] = ColorSpinorField(cuda_evec_param);
for (size_t i=0; i<nEv; i++) {
  quda_evec[0] = evec[i] ;
  #pragma unroll
  for( size_t n = 0 ; n < q.size() ; n++ ) {
    const size_t n1 = q[n].size() ;
    blas::block::caxpy(
         {coeffs[n].begin()+n1*i,coeffs[n].begin()+n1*(i+1)},
         {quda_evec[0]}, {q[n].begin(),q[n].end()} ) ;
  }
}
```

Here we call a block caxpy over iterators and read in only one evec when needed.

Cleanup and copies back to the host

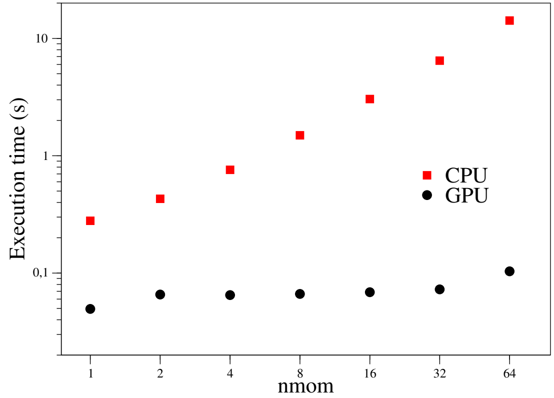
We of course must copy our object back to the host from the device

```
qudaMemcpy( return_array, d_ret, data_ret_bytes,
            qudaMemcpyDeviceToHost );
pool_device_free(d_ret);
pool_device_free(d_tmp);
pool_device_free(d_mom);
```

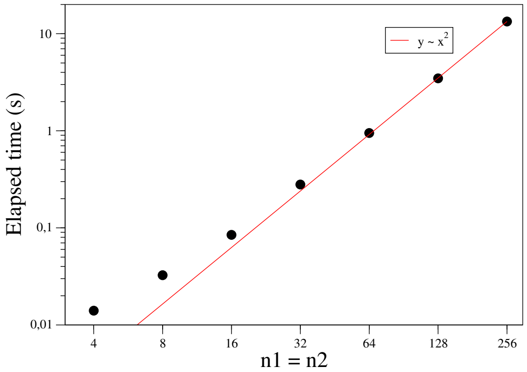
And free up the memory we have allocated

Note that we still aren't fully done yet as we have performed the DFT over the local dimensions only. It will be up to the end-user to finally put these correlators into global "t" indices. What we are returning is a small object so the burden on MPI shouldn't be bad.

Plots from Ernest



A comparison of naive single CPU vs GPU current kernels for a 16^4 local volume and $n1 = n2 = 16$ for various sizes of momenta. CPU scales linearly whereas GPU only starting to scale at large momenta.



GPU scaling with $n1 = n2$ noises, should scale like $n1^2$ and does at large $n1$ where the overhead in copies and things becomes negligible (blockSizeMomProj=16, nMom=32)

For our baryon contractions we will want to consider objects like the BaryonTriplet:

$$\begin{aligned} B(p,i,j,k,t) &= \sum_x e^{ip\cdot x}\epsilon_{ijk}\mathrm{Tr}_C[v_i(x,t)v_j(x,t)v_k(x,t)] \\ &= \sum_x e^{ip\cdot x}\mathrm{Tr}_C[(v_i \times v_j)\cdot v_k](x,t) \end{aligned} \tag{2}$$

Or the analogous BaryonKernel with the replacement $v_i \rightarrow \eta_i = \rho_{ij} v_j$. For the above version we only need compute index combinations $C(n,3)$ (for $nEv - 256$ this is a factor 6 saving).

We can write a kernel for the dotted trace (it is just the Lecture 2 one) and so we just need one for the cross product. These will be called `colorContract` and `colorCross` respectively.

Let's look now at the baryon kernel

```
void laphBaryonKernel( const int n1, const int n2, const int n3,
         const double _Complex *host_coeffs1,
         const double _Complex *host_coeffs2,
         const double _Complex *host_coeffs3,
         const double _Complex *host_mom,
         const int nEv,
         void **host_evec,
         QudaInvertParam inv_param,
         void *return_array,
         const int blockSizeMomProj,
         const int X[4] )
```

Much like the current kernel we have noise lengths $n1, n2, n3$ we also have 3 `host_coeffs`$=\rho_{ij}$ matrices that are the explicit noise matrices that will multiply the evecs with.

```
std::vector<std::vector<std::complex<double>>> coeffs(3) ;
std::vector<std::vector<ColorSpinorField>> quda_q(3) ;
quda_q[0].resize(n1) ; coeffs[0].resize( n1*nEv) ;
for(int i=0; i<n1; i++) {
  quda_q[0][i] = ColorSpinorField(cuda_q1_param) ;
  for( int j = 0 ; j < nEv ; j++ ) {
     coeffs[0][j*n1+i] = \
        (std::complex<double>)host_coeffs1[j+i*nEv] ;
  }
}
```

And likewise for coeffs2 and coeffs3, this is transposed so that dilution runs fastest. Note we are creating *n*1 noise-length evecs on the device for coeffs1 and so on for 2 and 3.

We can then call the blocked caxpy from before. This is not compute-intensive!

DFT parameters

Just like the meson setup but this time we will be ordered like $C(p, n1, n2, n3, t)$ with momenta moving slowest.

```
cublas_param_mom_sum.m = nMom;
cublas_param_mom_sum.n = X[3];
cublas_param_mom_sum.k = nSp;
cublas_param_mom_sum.lda = nSp;
cublas_param_mom_sum.ldb = nSp;
cublas_param_mom_sum.ldc = X[3]*n1*n2*n3;
cublas_param_mom_sum.a_stride = 0 ;
cublas_param_mom_sum.b_stride = nSites ;
cublas_param_mom_sum.c_stride = X[3] ;
cublas_param_mom_sum.batch_count = blockSizeMomProj;
```

The hot zone

```
for( int dil1=0; dil1<n1; dil1++ ) {
  for( int dil2=0; dil2<n2; dil2++ ) {
    colorCrossQuda(quda_q[0][dil1], quda_q[1][dil2], quda_diq);
    for (int dil3=0; dil3<n3; dil3++) {
     colorContractQuda(quda_diq, quda_q[2][dil3],
        (std::complex<double>*)d_tmp + nSites*nInBlock);
     nInBlock++;
     if (nInBlock == blockSizeMomProj ) {
       blas_lapack::native::stridedBatchGEMM(d_mom, d_tmp,
         (std::complex<double>*)d_ret + X[3]*blockStart,
         cublas_param_mom_sum, QUDA_CUDA_FIELD_LOCATION);
       blockStart += nInBlock ; nInBlock = 0;
      }}}}
```

Creates a "diquark" from the color cross product of q1 and q2. Color traces with q3 and DFTs the result.

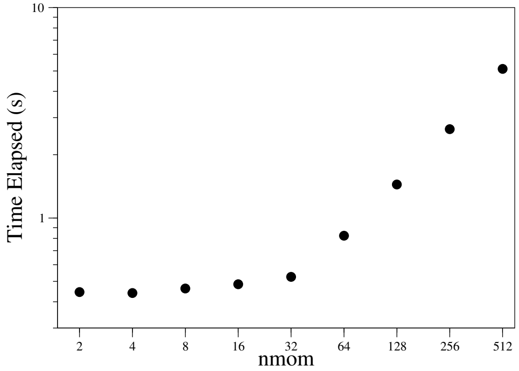
And finally the last copies and frees

```
qudaMemcpy( return_array, d_ret, data_ret_bytes,
            qudaMemcpyDeviceToHost );
pool_device_free(d_tmp);
pool_device_free(d_mom);
pool_device_free(d_ret);
```

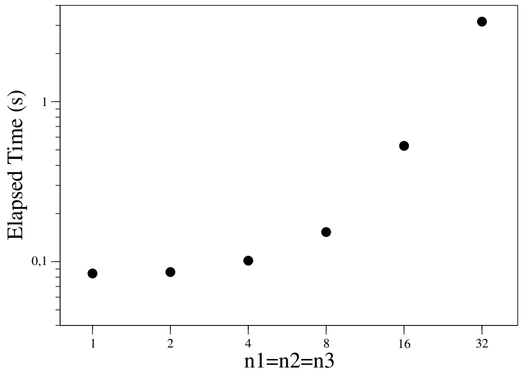
Of course the laphBaryonTripletA code works almost entirely the same way but without the coeff noise projection and a loop over C(nEv,3) instead of the flat $n1 \times n2 \times n3$.

Homework
Can you implement the colorContractQuda and colorCrossQuda kernels?

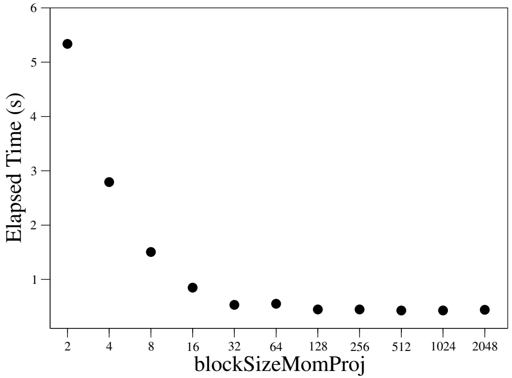
Check against:
https://github.com/RJHudspith/quda/blob/develop/lib/color_contract.cu
https://github.com/RJHudspith/quda/blob/develop/include/color_spinor.h



NeV = 64, $n1 = n2 = n3 = 16$ blockSizeMomProj=256, 16^4 local volume Only for many momenta do we start to scale - DFT is not where the time is spent until 64 momenta



NeV = 64, blockSizeMomProj=n1*n2, 16^4 local volume still not close to asymptotic n^3 scaling before I ran out of memory



NeV $= 64$, nmom $= 32$, $n1 = n2 = n3 = 16$, 16^4 local volume A factor of $10\times$ improvement from properly tuning this parameter

A note on memory usage for the laphBaryonKernel

d_tmp is of size (double complex)*V*blockSizeMomProj

ret is of size (double complex)*n1*n2*n3*T*nmom

q1, q2, q3 are size Nc*V*(n1/n2/n3) respectively

If our noises are large then we die. Usually ret is the problematic one and we could switch X[3]=1 and only work in 3D, however this is less efficient

Reducing memory footprint and accessing more flops per second really suggests we use single precision

To summarise

Current kernel with batching is fast and dominance against the CPU can be achieved by doing a large number of momenta and a large number of noises as this is where the good scaling kicks in and the overheads are better hidden.

The baryon kernel suits this as it is doing $n1 \times n2 \times n3$ traces and DFTs. For $n* < nEv$ a significant saving in memory can be achieved by just reading evecs in when needed and using blocked caxpy calls.

There is a sweet spot where the DFT is inexpensive and that is at $p \approx 64$ with multiple timeslices being used.

renwick.james.hudspith@gmail.com